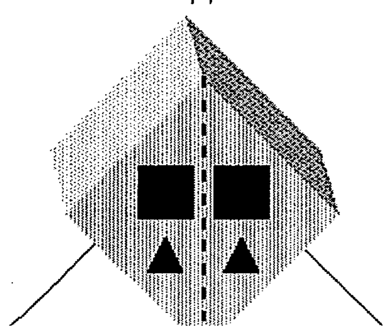
F26

Dr. Dobb's

Printer Friendly Version

Custom Application



CMP

Dr. Dobb's

HOME | ABOUT US | SUBSCRIBE TO DDJ | ADVERTISE WITH DDJ

GO TO...

Search

more search

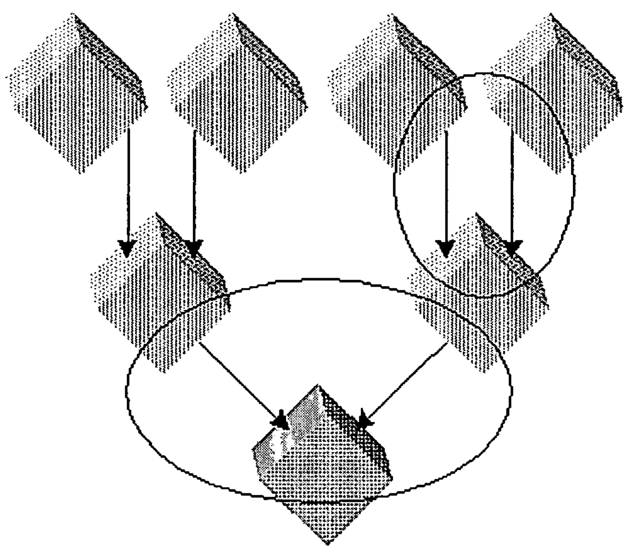
Logged in:
Martin W
My Account
Registration FAQ
687 Users Online
Today

Articles
Source Code
Newsletters
DevSearcher
Book Reviews
Embedded Systems
Software Careers
Sponsored Content
Dr. Dobb's Store
Subscriber Services
Premium Services

Dr. Dobb's Journal

Printer Friendly Version

Scalability   Namespace Domains



Client-Client-...-Server

HOME | ABOUT US | SUBSCRIBE TO DDJ | ADVERTISE WITH DDJ

SDMG Websites: BYTE.com, C/C++ Users Journal, Dr. Dobb's Journal, MSDN Magazine, Sys Admin, SD Exp
Unixreview, Windows Developer Network, New Architect

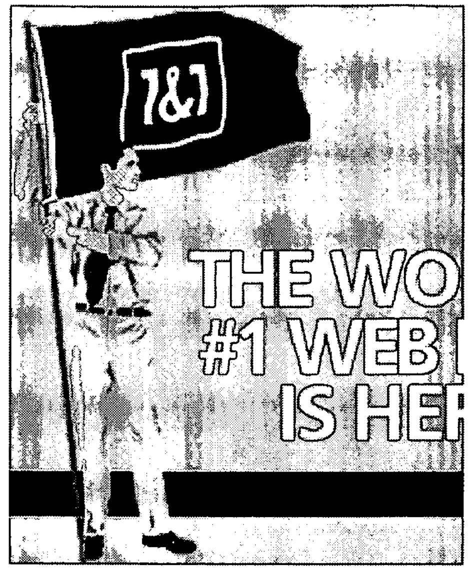
Dr. Dobb's

HOME | ABOUT US | SUBSCRIBE TO DDJ | ADVERTISE WITH DDJ GO TO...

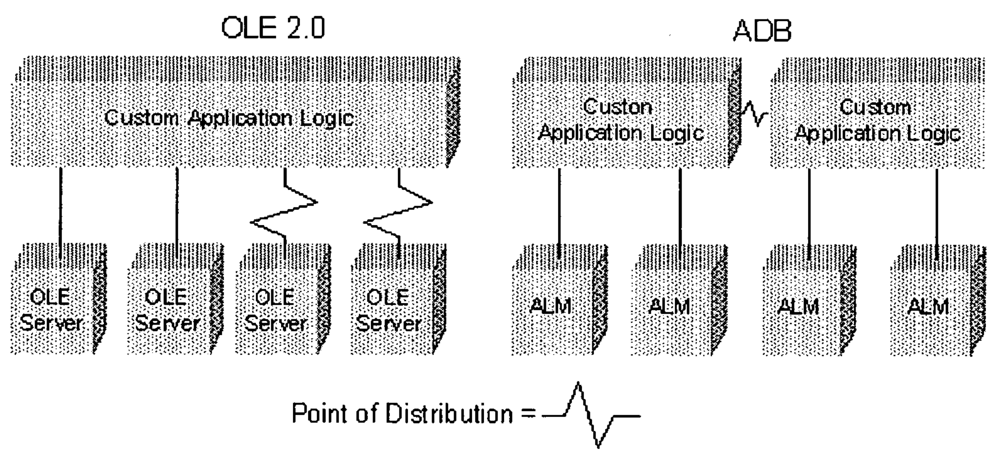
Search
more search

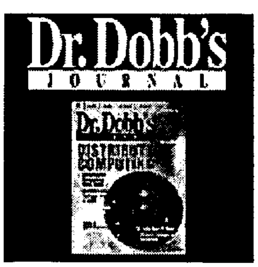
Logged in:
Martin W
My Account
Registration FAQ
687 Users Online
Today

Printer Friendly Version



ARTICLES
SOURCE CODE
NEWSLETTERS
DEVSEARCHER
BOOK REVIEWS
EMBEDDED SYSTEMS
SOFTWARE CAREERS
SPONSORED CONTENT
DR. DOBB'S STORE
SUBSCRIBER SERVICES
PREMIUM SERVICES





HOME | ABOUT US | SUBSCRIBE TO DDJ | ADVERTISE WITH DDJ


SPECIAL
BONUS
ISSUE
developer
from
Windows
Developer
Network
LANGUAGE
PERFORMANCE
BECHMARKING
Download
Today!

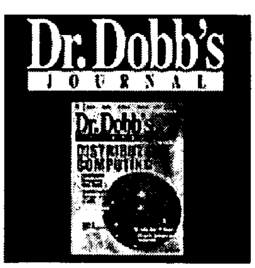

SPECIAL
BONUS
ISSUE
windows::
developer
NETWORK
Download
Today!

Dr. Dobb's

Search
more search

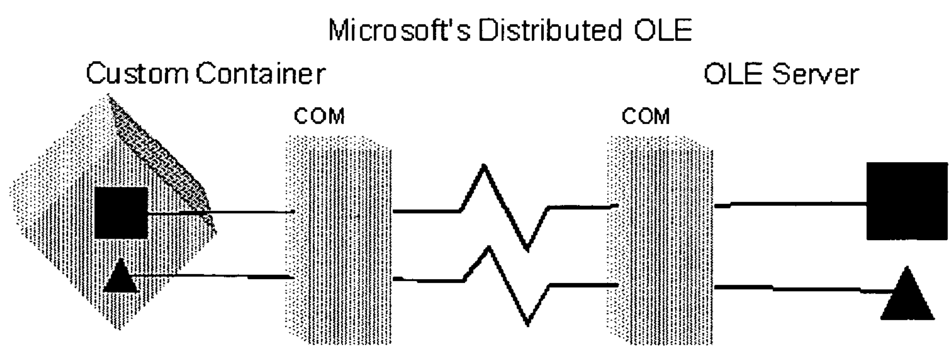
Logged in:
Martin W
My Account
Registration FAQ
687 Users Online
Today

ARTICLES
SOURCE CODE
NEWSLETTERS
DEVSEARCHER
BOOK REVIEWS
EMBEDDED SYSTEMS
SOFTWARE CAREERS
SPONSORED CONTENT
DR. DOBB'S STORE
SUBSCRIBER SERVICES
PREMIUM SERVICES



HOME | ABOUT US | SUBSCRIBE TO DDJ | ADVERTISE WITH DDJ
GO TO...

Printer Friendly Version



HOME | ABOUT US | SUBSCRIBE TO DDJ | ADVERTISE WITH DDJ


SPECIAL
BONUS
ISSUE
developer
Java
from
Windows
Developer
Network
LANGUAGE
PERFORMANCE
BECHMARKING
Download
Today!

Dr. Dobb's Software Tools for the Professional Programmer

Novell's AppWare Distributed Bus

Extending a powerful event engine across the network

Joseph Firmage

Joseph is vice president of Novell's NetWare Development Tools Division, including Novell's visual development tools: AppWare Bus, Visual AppBuilder, and AppWare Loadable Modules. He can be contacted at Novell Inc., 4001 South 700 E., Salt Lake City, UT 84107.

The software industry has been flooded in recent years with announcements of technology initiatives relating to distributed computing. Novell often refers to distributed applications as "network applications," because they leverage the power of the network in ways that stand-alone desktop applications can't. Examples of network applications include client-server databases, online services, e-mail applications, workflow software, transaction-processing applications, and remote-access applications.

These types of applications increase the productivity of workgroups of people, and often facilitate the replacement of aging terminals and mainframes with more-intelligent desktops and servers. Network applications now play a critical role in corporate information systems. These applications allow corporations to centrally locate shared services (as opposed to just shared data), while maintaining the advantages of intelligent, client-side applications running on today's modern distributed microcomputers.

Network applications are notoriously difficult to create with the tools commonly used for developing stand-alone desktop applications. A network application can consist of several pieces that operate on multiple computers as one overall software process. The developer of a network application faces an enormous integration task because different tools are used to implement different pieces of the application on different platforms with different operating systems. In addition to integration issues, the connectivity supporting the network application must support a broad range of network-transport technologies, address security and versioning requirements, and be maintainable (and quickly changeable) by user organizations. Perhaps as important as any of these considerations, such applications must be rapidly constructible.

Novell's AppWare is designed to resolve many of these difficulties. AppWare provides the tools and technologies to rapidly develop client applications that leverage existing network services, on multiple computing platforms. However, AppWare, as originally announced, did not offer a solution to those developers who required the ability to create all the parts of a network application--both the clients and the servers. Novell's AppWare Distributed Bus (ADB) provides this facility.

This article briefly describes the AppWare Bus and its distributed version, ADB. It then discusses such aspects of ADB as scalability, interoperability, service replication, Replaceable Transport Modules,

administration, and security. Finally, the ADB approach is compared with perceived alternatives, such as Microsoft's distributed OLE and OMG CORBA.

The AppWare Bus

The AppWare Bus is an event- and protocol-based communications and control engine for software components conforming to its straightforward API. It orchestrates the behavior of, and interaction between, its native components, called AppWare Loadable Modules (ALMs), and nonnative components, such as OLE and OpenDoc, by coordinating notification-based execution (true, queued-message-based invocation) within custom applications, which is particularly appropriate for distributed processing. Tools that contain the AppWare Bus are compatible with all ALMs and can offer to the programmer (or power user) the ability to link ALMs together to build custom applications. Novell's Visual AppBuilder is the first such tool. Other 5GL visual tools and 4GL scripting tools will be provided soon, working with the very same components and AppWare Bus.

Novell's ADB is a network-aware version of the AppWare Bus. ADB allows you to take any ALM-based application and partition part of the application on many clients, and the other part on one or more servers; see Figure 1.

The rationale for having a software bus is that all component conversations and behavior can be centrally managed, bringing all throughput under system control. ADB defines a model where the conversations between client and server logic are "atomically standard." That is, every conversation between the client and server is composed of objects whose data and control messages are the same, regardless of whether the software is running on the client or the server. Communicating between the two sides using these objects is, therefore, automatic and transparent to the application designer.

Today, this intercommunication is managed locally. To make the AppWare Bus distributed, it is necessary to reroute or redirect the event transmissions across a network. In so doing, the component parts of the overall application can work together automatically, transparent to the component creator. Component developers don't have to code to any distributed architecture or API. All existing ALMs support this distributed architecture because they use the AppWare Bus event engine. Application designers who use Visual AppBuilder (or any tool that incorporates the AppWare Bus) don't have to design their applications to be explicit clients or servers. They simply partition their projects, the distribution of which can be determined and changed at run time--application partitioning at its best.

An Example Application

As an example, say that your company is creating software for bank ATM machines and for back-end servers that process financial transactions from bankcard holders. This is a case where it is unreasonable to place the custom business logic that manages transactions in the client teller machine. Only the user-interface code should reside in the teller machine (along with the card reader and receipt printer), while the transactions are safely processed on a remote, secure, ultra-high-capacity server. When built using ADB, custom business logic can be rapidly developed to operate on servers. Desktop applications can share centralized functions (just as today they share files and databases) without giving up their own intelligent logic.

Users of Novell's AppBuilder may recall that an application project is organized into units called "subjects." To organize a large project, you break it into many subjects. To connect subjects together, you alias objects from one subject into another. The AppBuilder compiler (actually a part of the AppWare Bus) resolves the object aliases and produces a single application. With ADB, the user may

alias objects between projects themselves which, when compiled into applications, may be arbitrarily distributed on a network. The object aliases establish an interface that completely specifies the interaction between the application parts.

In this example scenario, two projects called "DB Client Project" and "DB Server Project" together comprise a client-server database application. The Do Query Button object is in the server project, with a function chain connected to it that will be executed when the button is "Pressed." An alias of the Do Query button is placed in the client project, along with a Window object to display it. This button alias implicitly establishes an interface between these two projects. After compilation, the resulting application partitions can be placed together or apart. When the client user presses Do Query, the function will execute in the server, wherever the server is. Similarly, simple text fields, pictures, not-so-simple tables, or any other ALM object instance that contains data (or controls something) can be aliased. When such instances experience a change in their data, this change is reflected in their counterparts, wherever they may be on the network. If an instance issues a control event, this will trigger any responses connected to counterparts on the network.

The client or server partitions can be changed or replaced at any time, simply by maintaining conformance to the object alias interface. This allows clients or services to grow and to offer new features without breaking existing clients or services. It is a simple and effective concept, though impossible without the notification-based execution model provided by the AppWare Bus.

How ADB Works

As discussed previously, the AppWare Bus employs a notification-based processing model for executing applications. ADB extends this model by placing a network connection in the central processing queue. The network connection itself is DLL-replaceable by Novell or third parties. It can be a simple RPC, ORB, pipe, or any other network namespace or transport engine. Within an application partition, ADB intercepts data and control events for object-alias instances that reference original objects of another partition. ADB reroutes and replicates these events through the network connection as needed for distributed data and control.

More specifically, ADB monitors the event queue for the following events:

- *ObjectChanged* data events, carrying affected object data along.
- Object signals, triggering custom logic in any partition where function chains exists for them.
- Object event protocols, for inter-ALM conversations underneath higher application logic.

Duplicates of these events, along with affected object data, are passed to object aliases, or to originals in other partitions of the overall client-server network application.

Once an event is rerouted, the same ALM code is operating on the receiving end to process the event. So, it is impossible for the message being transmitted not to be understood. No agreement by committee (or otherwise) must be reached as to the format of the information crossing the wire, since the sender and recipient are, in fact, the same type of ALM.

Note that ADB can create a distributed application around any component, not just those defined by Novell. As long as the component employs the AppWare Bus mechanisms for signals and data management, ADB can distribute the processing transparently. Novell will provide the wrappers for OLE and OpenDoc components, so that the AppWare Bus can control and leverage them as well as ALMs. Thus, when you create an ALM, you are, in effect, defining a new object that becomes a de facto

network atom whose data and control are understood by any other foreign application client or service created with the same type of component. By such event replication and routing, along with service replication (described in a later section), the process of building a distributed application becomes simple, but remains powerful.

If so designated by the application designer, a client or service partition may be queried at run time (by way of an administrative ALM described later) for its object-alias-interface specification. The response to such a query is in the form of a table containing the portion of the interface designated as public by the application designer.

Examples 1 and 2 present the object-alias interfaces for the client and server partitions of the example banking application. (Although, in such a situation, you probably wouldn't make the application interfaces public!) In this application, the server provides original objects to transfer information (and requests) to/from the teller machine. The teller machine provides alias objects to transfer information (and requests) to/from the server. Notice that the server's interface title, "Bank of Novell," is called the service type. It is this service-type identifier (not the individual objects) that is registered in the namespace on the network. Each instance of a service of a given service type has a service name (which can be any string) giving a proper name to that service instance for network registration. Consider the following analogy. A workgroup might have two laser printers, one named "Engineering" and the other named "Documentation." Both network entities are of the type "LaserWriter." It is the same with service names and service types.

At design time, the partitions can be assigned version numbers and compatibility numbers, so that clients and servers can both be improved and changed over time. Every client and server indicates its version numbers, and the server indicates its minimum compatibility version for clients. ADB will not allow a connection to be made between a client whose Required Server Version number is lower than the server's Server Compatibility Threshold, or higher than the server's Server Interface Version. In this example, note that the Bank of Novell server offers a picture object ("Advrtsmt") to display as an advertisement while the cardholder is waiting for the transaction, but the Teller Screen client doesn't support that facility (perhaps because that particular teller has just a text-only display ability). The server is version 2, but remains compatible with clients expecting version 1 or 2.

The "Object" type column contains the 32-bit unique ID registered with Novell for your ALM object type when you created it. ADB does not understand the type reference. It's there to match types with counterpart objects in counterpart interfaces. The type "Text" doesn't mean anything to ADB, but ADB can match that ID with IDs in counterpart interfaces. Registering type IDs with Novell assures their uniqueness.

Each object in the interface may designate whether it can receive and/or send information. ADB thus gives the application designer the ability to restrict the flow of data and control to given directions. When an object can both send and receive information, ADB enables special logic to ensure that data and control "echoes" do not occur.

Note that the client interface is headed by the words "Interface of 'B of N Teller Screen' for 'Bank of Novell.'" A given client partition may simultaneously be a client to more than one server, and may have other object-alias interfaces that refer to other server interfaces. Thus, a request to return a client's object-alias interface must specify the server to which the interface applies.

Service Replication

One design challenge common to most client-server system architectures is enabling many client applications to simultaneously access a common service. Transparently separating one large application into two pieces (one of which can run on a desktop and the other on the server) is practically useless. Transparently separating one large application so that one of its pieces can be used on many desktops while the other piece is deployed on one server is *very* useful. In short, one-to-one distributed processing is not client-server in nature. Many-to-one distributed processing is enormously valuable and directly fulfills the goals of the client-server paradigm for all application processes, not just SQL databases.

Some people claim that it is as simple as taking a typical application-level C++ class and using its interface declaration on the client to call the implementation on a server. In practice, however, this often delivers a system that is sound only when there is one client. In order for many clients to connect to a single server, the implementation on the server must anticipate the fact that many clients may use its services simultaneously. In fact, it is very difficult to create code that handles many client requests simultaneously.

While this issue can be addressed by properly designing and developing the service, doing so has usually required the talents of highly trained programmers. Since the fundamental value of AppWare is that it enables business programmers and knowledge workers to create network applications (and, with ADB, create the custom-network services, as well), it is necessary to transparently incorporate this sophisticated, many-to-one service logic into the AppWare Distributed Bus. ADB has a facility called "service spawning" that addresses this problem.

Service spawning allows you to designate a server application partition that will replicate a portion of its data space and optionally create a new task for each client that connects to it. Thus, when a client initiates a connection to a server, the server may automatically extend itself, so that the server partition can maintain state information for the client. Given this facility, server logic can be designed as if only one client were connected to the server.

Other partitions of the same overall application may not spawn, perhaps serving the spawning partitions once the information from their clients has been reduced to a stateless form (such as one-way transactions to/from a database). In fact, if you wish, servers need not spawn at all. You are then responsible for assuring the integrity of multiple client-server conversations. Generally, however, spawning server partitions simply centralize portions of multiple client applications that would otherwise consume the same computing resources on lesser-powered, distributed desktops. By centralizing what would otherwise run on clients, you can improve performance as a shared function is brought into close proximity with shared data. In addition, the clients are insulated from implementation and any changes to it. Further, the client applications themselves can grow independently from the server applications, retaining whatever logic and intelligence with which you may empower them.

Replaceable Transport Module

Neither ADB nor ALMs depend directly on any one network transport facility to accomplish distributed processing. ADB specifies an API that it invokes to fulfill the namespace and transport requirements underlying distributed applications. Thus, Novell and third parties are free to replace the Novell-supplied namespace and transport simply by replacing a particular DLL with one containing an alternative implementation. Such alternatives might include RPCs, ORBs, or even simple, direct serial communication links. The scalability, flexibility, mobility, communications performance, and network-protocol support for ADB distributed applications are all determined by the Replaceable Transport Module (RTM).

An RTM must provide the following essential abilities:

- Register network-visible entities in a namespace under a certain type and name string.
- Respond to name queries for all network-visible entities of a certain type and/or name.
- Open extended conversations (for example, not single transaction) between two named entities.
- Open reasonable numbers of simultaneous conversations between two such entities.
- Make intranode connections transparent if local deployment is to be supported.
- Deliver and receive arbitrarily large data streams, reliably or with robust error detection.
- Provide asynchronous callback hooks for send-complete/receive-started occurrences.
- Abstract the transport protocol, if necessary.

The RTM can provide user connection, authentication, security, and encryption facilities itself, but this is not required. It is generally presumed that some basic level of such services will be supplied by the underlying network operating system (NOS) in an enterprise environment. Tight security at the level of application logic may be employed by using the administrative ALMs.

Novell will supply an RTM for ADB, bundled with AppBuilder Version 2.0. This RTM is scalable for use in large enterprises by way of a dynamic namespace called NetWare Directory Services with multilevel domains and dynamic replication avoiding single-point-of-failure. The RTM can transport over IPX/SPX, TCP/IP, or AppleTalk, one at a time (interprotocol transparency is optionally available), and uses the underlying network operating system (NOS) for connection, authentication, and other security provisions.

Novell will also provide an RTM based upon the well-known Tuxedo transaction processing system. This RTM uses a transaction-based communication model that provides dynamic load balancing for use in large-scale deployments.

Novell also will supply an RTM for a simple serial connection, for use in dial-up modem access to non-LAN-based online services built upon ADB. This RTM provides a serial transport facility with essentially no namespace--the namespace is limited to the caller and the callee. This RTM is important because the performance and reliability benefits of ADB are most clearly seen in the context of a slow, unreliable connection. ADB can thus facilitate the building of online services.

An RTM can be replaced at any time, without even requiring recompilation of ADB-based applications. However, note that distributed applications running on different RTMs are not automatically interoperable; to attain interoperability, gateways must connect namespaces and transport formats.

Heterogeneous Interoperability

Novell has defined a specification for the way ALM objects import and export their attributes to/from a universal, atomic data expression. This expression consists of data types such as strings, numeric values, images, sounds, arrays, and so on. The import/export facility processes data in Universal Program Structure (UPS) format. Historically, UPS has been used to port ALM-based application projects between different desktop platforms (such as Microsoft Windows and Macintosh).

ADB uses UPS to provide rapid run-time data conversion for object values moving between different native OS platforms. This conversion may involve more than just byte-swapping. For example, the Picture ALM on Macintosh computers uses the PICT standard to display images, while on Microsoft Windows the Picture ALM uses BMPs and metafiles. So, the implementation of UPS on Macintosh can convert between a Mac PICT and a UPS image, and the Microsoft Windows implementation of UPS can

convert between a BMP or metafile and a UPS image. The Picture ALM supports the import/export entry points necessary to interact with UPS. A Picture object can thus be moved anywhere that UPS is implemented. The implementation of UPS on each given platform supports conversion between that platform's common OS data types (as well as simple data types) to UPS universal equivalents, so that the ALMs themselves need only have the ability to port native object data to UPS types, rather than every analogous type supported by any target platform.

When ADB connects two partitions, each running on the same native operating system, no UPS conversion takes place, and, thus, native information is transmitted.

Scalability

The scalability of ADB-based distributed applications depends on the ability to construct multilayered, client-server network applications. ADB allows any number of hierarchical layers of clients and servers. This means that a client partition can itself be a server to a higher-level client.

Scalability also depends on the namespace employed in the RTM. Consider a client-server application with 10,000 users and 200 servers in a large corporation. If, as each user launches the client application, all 200 servers were visible and accessible, it would be very difficult to navigate and manage resources, connections, and network activity. The network traffic alone required to dynamically display the names of all 200 servers to each of the 10,000 users would overload any common NOS. The namespace is responsible for more intelligently setting hierarchical contexts that restrain the view and access of users to other users and servers; see Figure 2.

The RTM included with ADB includes a comprehensive namespace technology that can automatically (and under administrative control) establish multilevel domains imposing the necessary scopes to prevent unmanageable situations from arising. This RTM can accommodate unlimited numbers of concurrent users by using these namespace domains.

Reliability and Performance

One key attribute of ADB unmatched in most other distributed object models is that there is no redirection of OS or component-code routines. "Redirection" is the process whereby a routine is fooled into thinking that it is running locally when it is actually running on a remote machine. File and printing services in all common network operating systems are implemented in this fashion (the file and print APIs of the local operating system are redirected toward remote devices). Microsoft's stated plans for distributed OLE also follow this general model (OLE server interfaces are redirected to remote implementations). In theory, all applications that use those local interfaces transparently will get the benefit of the shared resource.

The upside of redirection is that it automatically works with existing applications. One of the most serious downsides is the difficulty of handling error conditions that occur when network connections are broken or otherwise fail. For example, when a Read operation fails on a server disk drive as a result of a communication problem, the number of levels requiring perfect error trapping in the user's OS, objects running on the OS, and in the custom code of the application, is extremely large.

In applications based on ADB, the connection is not established by the redirection of code routines executing in the OS or in the ALM components. Rather, the connections between different segments of the application logic made by the application designer are redirected across the wire by the AppWare Bus. So, if a failure in the network connection occurs, the worst that can happen is that your particular

application partition triggers an error condition requiring some handling in your client and/or service-application logic. You must incorporate reasonable error trapping in your logic to handle for this type of occurrence (which you'd have to do in any distributed model). ALMs are provided to assist in detection. In the ADB model, it is impossible for a network-connection failure to result in a crash or system fault in the OS or ALMs, regardless of whether ALM developers implement perfect error checking--there is no explicit or implicit networking in ALMs around which to error-check. This key reliability benefit is not well recognized, even in the evolving era of mobile computing in which stable network connections will be rare.

The same architectural issue drives performance. In general, as the network conversation becomes less frequent and more coarse-grained, the overall application performance dramatically rises, and the ratio of overhead to data becomes insignificant. ADB's architecture will provide one of the highest performing models for distributed processing, approaching the architectural performance of hard-coded transaction-processing systems.

Administration and Security

Along with ADB itself, Novell will supply three ALMs that allow you to include administration and security functionality in server and client projects. These ALMs also can be used to create separate management applications. The three ALMs are Client, Server, and Monitor. Table 1 summarizes the functions and signals in these three ALMs.

As you can see in Table 1(a), the Client ALM provides functions to manually connect and disconnect to and from a specified server partition. The server's identity is either known in advance or discovered by way of functions that list services available in the client's network context. Client partitions can be designated at compile time to connect automatically--a setting that can be overridden by the server partition for authentication purposes. The Client ALM also provides functions to list the types of servers available to connect to, and individual servers of each type.

The Server ALM shown in Table 1(b) allows the server's logic to authenticate client login requests using whatever application logic it wants (including calling the NDS ALM to check in with the global NetWare directory) and accept or refuse such connection requests. The ALM also allows server to establish an exclusive conversation with a particular client, temporarily locking out other clients. Finally, the Server ALM provides functions to disconnect clients and list all connected clients.

The Monitor ALM, summarized in Tables 1(c) and 1(d), allows both the client and the server logic to describe the object alias interface of the counterpart's partition, temporarily suspend the transmission of information for an object or the whole interface, and manually refresh object values in counterpart partitions. The Monitor ALM also offers an object that provides several signals, triggering client or server functionality when important connection-related events occur.

Remember that the implementation of many of these functions is supplied by the RTM. The AppWare Bus abstracts away differences between RTM implementations, so that applications don't need to be recompiled to operate on a new distributed namespace and transport. Note also that all these functions inherit the administration, authentication, and security provisions of both the RTM and the underlying NOS network connection, if there is one.

Other Distributed Object Models: OLE and CORBA

Figure 3 illustrates the fundamental contrast between distributed OLE and ADB. At this writing, OLE

2.0 is not fully enabled for distributed applications. Microsoft plans to make OLE 2.0 distributed by inserting a general RPC in its Component Object Model (COM), the underlying object library engine. This is planned to be released in conjunction with Cairo in 1996. These relationships are shown in Figure 4. The key question is where the transparent distribution occurs. As we understand Microsoft's plans, OLE will distribute processing at the seam between the custom application logic and OLE server components. Thus, when an OLE 2.0 container application invokes an OLE 2.0 server through COM, the server itself could be executing remotely on Cairo, connected via RPC.

Transparent distribution can't really occur within custom application logic unless there is some system--like the AppWare Bus--orchestrating the execution of the application logic. AppWare Distributed Bus thus can distribute at one or more arbitrary seams within the custom application logic. ADB places control over the distribution seam(s) in the hands of the application builder, who does not need to know the complexities of crafting a service in the form of an object. "Partitioning" is targeted at the easy separation and distribution of portions of application logic, right within a single overall application logic, rather than the creation of a separate service encapsulated as an object. For those who wish to take the time to create services as objects, AppWare's support of OLE and the other object models will facilitate the use and eventual creation of these components as well, so users get the best of both worlds.

CORBA is a technology initiative led by the Object Management Group (OMG). OMG's goal is to establish agreement among major system and application software vendors on a universal object-oriented model for expressing interfaces for distributed services (instructions on how to access network services), and for accessing and using such interfaces to execute distributed services. Such a system is called an "Object Request Broker" (ORB). By achieving agreement among the various vendors, network services become interoperable and interchangeable.

An ORB is a general abstraction for sophisticated developers to use in writing client/server software systems, where services are encapsulated as universally usable objects. ORBs will end up working underneath several higher-level network middleware and application software systems, including partitioning systems. An ORB can therefore be used to create the Replaceable Transport Module used by ADB to fulfill distributed processing. Thus, given an RTM based on a CORBA-compliant ORB, ADB-based applications are CORBA compliant. Novell expects to provide an RTM based on a CORBA-compliant ORB.

Of course, AppBuilder's forthcoming support of a number of key object models will guarantee that services exposed as CORBA-compliant objects will be fully usable, and eventually can be created, within AppBuilder.

Conclusion

The ADB provides a fundamentally different approach to distributed applications. It approaches the problem by setting three fundamental objectives. Custom application logic must be able to be placed on the server. A "periodic table of software elements" comprising a universal expression of atomic data and control between local and remote applications must be supported. The system must not demand synchronous execution.

In Novell's vision of the future of network computing, AppWare will play an important role, and ADB will provide a remarkably simple and powerful way for AppWare users to leverage the power of distributed processing. It has immediate relevance to any business that wants user applications share functions as that business shares data data today, within the context of distributed networks of microcomputers.

Figure 1 AppWare distributed bus (ADB) allows partitioning an application between client and server. Figure 2 Namespaces provide scalability by limiting visibility. Figure 3 Comparing distributed OLE with ADB. Figure 4 The structure of distributed OLE.

Example 1: Server interface for bank ATM application.

```
Interface of "Bank of Novell"
Server Interface Version: 2
Server Compatibility Threshold: 1
```

Object	TypeID	Kind	Direction	Comment
Bank Key	Text	Original	Receive	Bank's key to get into system
Card ID	Text	Original	Receive	Card holder's card #
PIN	Text	Original	Receive	Card holder's PIN #
OpCode	Nmbr	Original	Receive	Transaction selection
Amount	Nmbr	Original	Receive	Dollar value of transaction
Confirm	Subr	Original	Receive	Confirm action to Message
Cancel	Subr	Original	Receive	Cancel action to Message
Dispense	Subr	Original	Send	Dispense cash
Message	Text	Original	Send	Content sent to teller screen
Receipt	Text	Original	Send	Content sent to receipt printer
Advrtsmt	Pctr	Original	Send	Ad to display while waiting

Example 2: Client interface for bank ATM application.

```
Interface of "B of N Teller Screen" for "Bank of Novell"
Client Interface Version: 1
Required Server Version: 1
```

Object	TypeID	Kind	Direction	Comment
Bank Key	Text	Alias	Send	Bank's key to get into system
Card ID	Text	Alias	Send	Card holder's card #
PIN	Text	Alias	Send	Card holder's PIN #
OpCode	Nmbr	Alias	Send	Transaction selection
Amount	Nmbr	Alias	Send	Dollar value of transaction
Confirm	Subr	Alias	Send	Confirm action to Message
Cancel	Subr	Alias	Send	Cancel action to Message
Dispense	Subr	Alias	Receive	Dispense cash
Message	Text	Alias	Receive	Content sent to teller screen
Receipt	Text	Alias	Receive	Content sent to receipt printer

Table 1: The C/S column denotes whether the function or signal applies to a client (C) or server (S) partition. The I/O column denotes whether the function or signal applies to the set of objects comprising the interface (I) or just an individual object (O) in the interface. (a) Client ALM

functions; (b) Server ALM functions; (c) Monitor ALM functions; (d) Monitor ALM signals.

```
(a)   C/S   I/O   Functions

      C     I     Connect
      C     I     Disconnect
      C     I     Is connection
      C     I     List service types
      C     I     List services
      C     I     Get client ID

(b)   C/S   I/O   Functions

      S     I     List clients
      S     I     Start exclusive
      S     I     Stop exclusive
      S     I     Disconnect client
      S     I     Authenticate client
      S     I     Accept client
      S     I     Refuse client

(c)   C/S   I/O   Functions

      C/S   I     Describe interface
      C/S   I/O   Suspend
      C/S   I/O   Resume
      C/S   I/O   Refresh

(d)   C/S   Signals

      C/S   Connection requested
      C/S   Connection accepted
      C/S   Connection refused
      C/S   Connection closed
      C/S   Connection broken
      C     Auto-connect failed
```
